

Liberty FSE Capstone Project - Project Overview & Work Steps

Overview

This capstone project involves using React, NodeJS, Express and MongoDB to build an end-to-end "News Reader" application. The application allows users to create, submit and updated queries that return a list of news article along with links to those articles. The front-end of the application is a web application built with React. The back-end of the application consists of a REST service that manages requests for data from a MongoDB database, news queries that are passed through to an online REST api at newsapi.org and application state data that is saved and retrieved from disk by the REST service itself.

Here are some high-level points to remember about the project:

- You are required to construct the front-end application using React.
- You are required to construct the back-end applicaiton using NodeJS, Express and MongoDB.
- The back end application will make calls to a REST api at "newsapi.org". Calling this service requires an apiKey that you will obtain early on in the project by signing up for a free account at "newsapi.org"
- The end-to-end application should be functionally similar to the provided Demonstration application.
- You will create the application from scratch and add features according to the provided requirements.
- Development steps to follow are suggested later in this document.

Development Environment

The project development environment includes:

- Windows, Mac or Linux PC
- Chrome web browser
- Internet Access
- Visual Studio Code Editor/IDE
- NodeJS
- Git

You can use the supplied lab machines or your own computer if you already have a development environment that meets the above specifications.

The Project Zip

A zip file named: **WA3484-project-student.zip** containing the files listed below will be provided. The zip file should be copied to your development machine and unzipped in a convenient location. *Make a note for yourself of the location as you will be needing it later on in the project.*

The Project Zip contains these files and directories:

Filename	Description
Project-Overview.pdf	This file
Setting-Up-the-Demo.pdf	Instructions for setting up the e2e project demo
/ProjectAssets	Directory containing files and directories referenced in this document as well as project demo and solution files.

NewsApi.org Access

The core of your end-to-end application will be its ability to display news article lists that are obtained by calling the newsapi.org REST service.

In working with the newsapi.org API you will be following a common pattern that many real-world projects do. That is, the data you need to interact with will come from an existing API server over which you have little or no control. Data interactions are managed by introducing a REST API of your own which is responsible for calling the existing back-end service.

Newsapi.org consists of a web site and REST api which share the following URL:

<https://newsapi.org/>

The newsapi.org web site includes documentation on how to use the newsapi.org REST API service. Signing up to the web site will provide you with the apiKey required to call the newsapi.org REST service.

You should complete this sign-up before going any further and save the apiKey in your notes. You will need the apiKey to run the end-to-end application you create as well as to run the project demo.

Registration link for newsapi.org site:

<https://newsapi.org/register>

The Demonstration Project

The demonstration project is a full implementation of the application you are building. By running the demo you will see one specific implementation of the project requirements. A zip file containing the project demo files is located in the ProjectAssets directory.

ProjectAssets\e2e-demo.zip

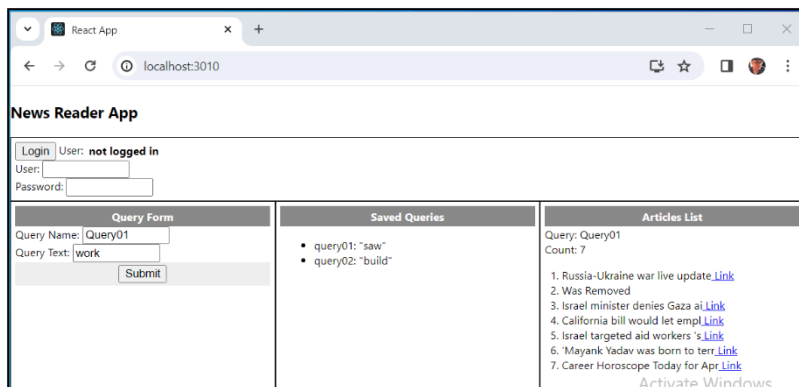
Instructions for unzipping the file and setting up the various demo components is available in the WA3484-project-student.zip:

{wa3484-project-student.zip}\Setting-Up-the-Demo.pdf

Go ahead and follow these instructions now to setup the demo.

Note: You will need the apiKey you got earlier when you registered at newsapi.org

The application that comes up should look like this after clicking on the "query02" saved query.



Try out the application. To login use either of the following credentials:

admin/admin
guest/guest

The login capability is used to differentiate the app's features for various types of users. For example:

- Existing queries can be selected but no new queries can be created unless a user is logged in
- "guest" users can create up to three queries with limited properties
- "admin" users can create unlimited queries with additional properties

Please note that the ports used by the demo differ from the ones you will be developing with. This will allow you to keep the demo running while you are developing your own implementation:

DEMO Application Ports

- Port 4010 - Back-End server
- Port 3010: Front-End application

PROJECT Application Ports

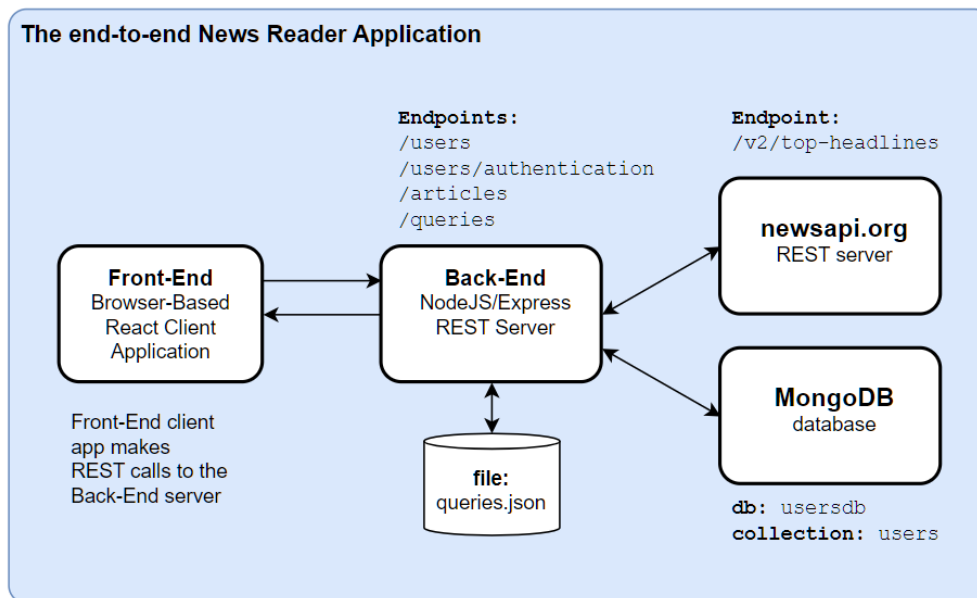
- Port 4000 - Back-End server
- Port 3000: Front-End application

The Project Solution

Project solution repositories are available and serve a different purpose than the demo application. The demo application can be used to see how the app is supposed to function. The solution code on the other hand has these uses:

- Solution code can be inspected for ideas on how to proceed with your own coding
- If you run into trouble with your own implementation you can build on top of a given solution

Solution Overview Diagram



The Solution Code

The solution code comes in the form of a zipped up repositories which are available here:

ProjectAssets\front-end-solutions.zip – solutions for the React front-end application
ProjectAssets\back-end-solutions.zip – solutions for the NodeJS back-end server application

The solution repositories have been tagged to allow checking out, inspecting and running code at various stages of development.

Example setup: front-end solutions repository:

1. Unzip the front-end-solutions.zip file into your C:\ProjectWork directory. You should end up with the following: C:\ProjectWork\front-end-solutions.
2. Open a terminal
3. Navigate to the project directory: **C:\ProjectWork\front-end-solutions**
4. Run "npm install" in the project directory
5. Run "npm run start" to start the front-end

Notes on setup of the back-end solutions:

1. After unzipping back-end-solutions.zip you will need to start both the mongodb server and the back-end server
2. To start mongodb run: npm run mongodb
3. To start the back-end server run: npm run start

Execute the following in the project directory to see a list of the available tags:

```
git tag
```

By default the repository's working directory contents correspond to the final solution tag (the one with the highest number).

Checking out solutions:

One solution can be checked out at a time.

Specific solutions are checked out like this:

```
git checkout {tag-name}
```

example:

```
git checkout Stage02-03
```

If you want to checkout a solution tag and the server is already running, shut down the server first and then restart it again after checking out the solution tag.

Tags Available in the **BACK-END Server** project

Tag Name	Project Stage Description
Stage02-01	Initial server with dummy endpoints
Stage02-02	News GET endpoint working
Stage02-03	News POST endpoint working
Stage02-04	Users GET endpoint working
Stage02-05	Users/Authenticate POST endpoint working
Stage02-06	Queries POST & GET endpoints working

Tags Available in the **FRONT-END Client** project

Tag Name	Project Stage Description
Stage02-01	Initial server with dummy endpoints
Stage02-02	News GET endpoint working
Stage02-03	News POST endpoint working
Stage02-04	Users GET endpoint working
Stage02-05	Users/Authenticate POST endpoint working
Stage02-06	Queries POST & GET endpoints working

Running project solutions while developing:

If you plan to run the solution code at the same time as you are developing and running your own implementation then it is a good idea to change the port numbers of the solution to something other than what you are using in development.

For the React app update the script in **package.json** to specify a different port - like this:

```
"start": "set PORT=3100 && react-scripts start",
```

For the back-end server app you can set the port on the following line of the **server.js** file:

```
const port = process.env.PORT || 4100;
```

Final notes about the project solutions:

Whenever you have a question about how your application should work you can refer back to the project solution.

Although your application should work like the solution does, your application code may not exactly match that of the solution. Where differences occur you should be able to explain how why you coded the way you did.

Project Stage Overview

Work on the application should proceed in the following order:

- Stage01 – Getting Started: Read and understand provided materials
- Stage02 – Create the back-end REST API: Test it with postman
- Stage03 – Create the front-end React Client Application
- Stage04 – Add Optional Application Features
- State05 – Clean-up and Prepare for Presentation

Project Work Steps

This section includes suggested steps for implementing your application. The suggestions are organized by stage.

Stage01 – Getting Started

In this stage you should complete the following:

- Read and get familiar with the provided project documentation and files,
- Setup the demo application,
- Get to know the restapi.org API and documentation,
- Work with the restapi.org REST API directly using postman

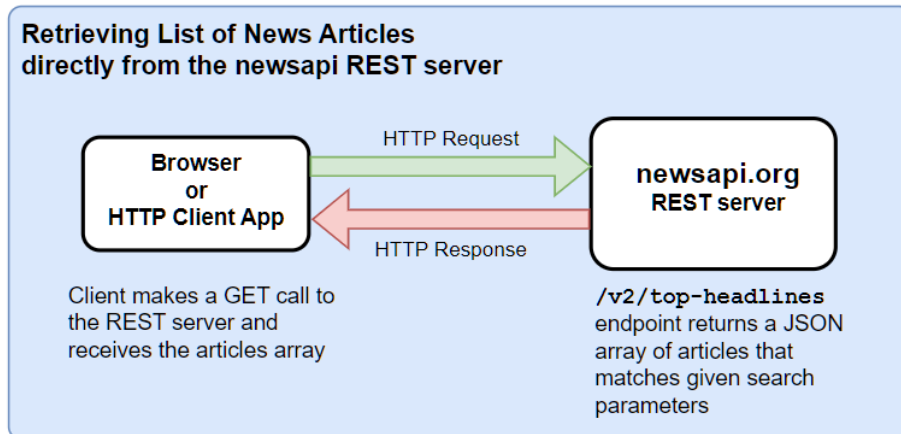
When you are done with this stage you should be ready to start development on the back-end server.

Getting to know newsapi.org

By this point you should already have, registered with newsapi.org and obtained an access token. If not then go back and do that now. Instructions appear earlier in this document.

newsapi.org maintains two API endpoints. The one you will be using for your application is the "Top Headlines" endpoint.

`https://newsapi.org/v2/top-headlines`



The newsapi.org Documentation

The documentation includes a section on custom newsapi client libraries. You will not be using these. Instead you will be make various HTTP requests using the JavaScript `fetch()` function.

Read through the following sections of the newsapi.org website:

<https://newsapi.org/docs/get-started#top-headlines>

<https://newsapi.org/docs/endpoints>

<https://newsapi.org/docs/endpoints/top-headlines>

<https://newsapi.org/docs/errors>

As the documentation states, newsapi.org API requests consist of GET requests with additional data supplied through query strings appended to the URL, like this:

```
https://newsapi.org/v2/top-headlines?q=news&country=us&apiKey=b24f9d5564944ab3acd556097cece75x
```

These requests work over http as well as over https.

Newsapi.org Free Account Restrictions

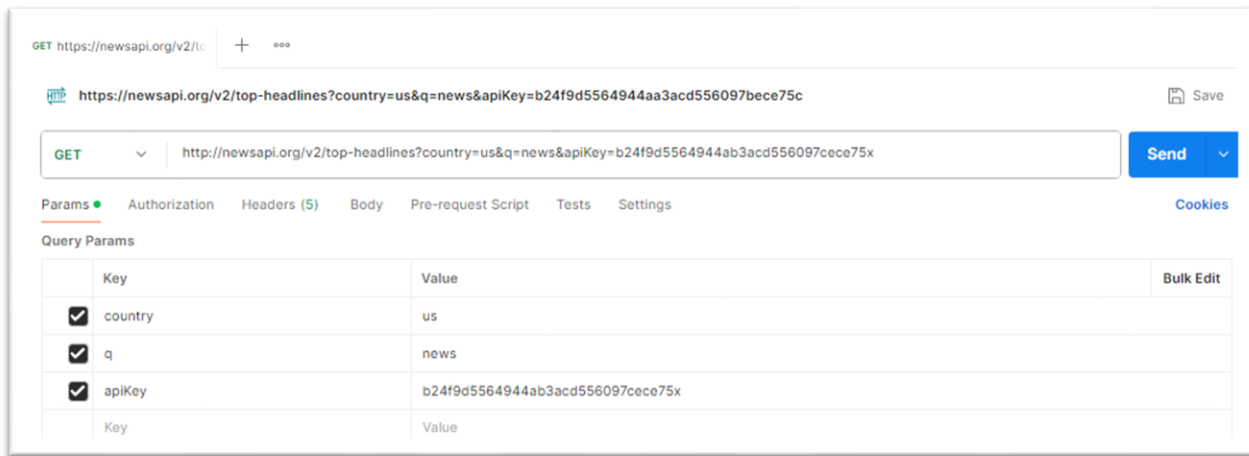
newsapi.org restricts the number of requests per day for free accounts. Though they state a limit of 100 queries it appears that they are sometimes flexible and allow more. If you go over the limit for the day you will start receiving error message replies for all your API calls (until the next day). That being said, it should be possible to go about your development without running into this issue. Just be careful and don't make lots of unnecessary requests.

Creating Requests with Postman

To create the URL with query parameters in Postman you can use the "Params" tab. It allows you to enter each query paramter key/value pair on a separate line. Postman then constructs the query parameter string and adds it to the URL automatically.

Liberty FSE Capstone Project - Project Overview & Work Steps

Page 9



Try submitting some requests to the newsapi.org api. For each request use at least the following parameters:

- q – query text
- country – language
- apiKey – you api key for newsapi.org
- pageSize – how many results should be returned

Once you feel that you understand how the newsapi.org API works you can move on to the next stage.

Stage02 – Create the back-end REST API

In this stage you will create a NodeJS/Express server with endpoints that provide access to:

- News article listings
- Saved query lists
- User records

The REST server should include these endpoints:

Endpoint	Used-By	Purpose
/news GET	web browser	Used to test the connection to newsapi.org server from browser
/news POST	front-end app	Used to retrieve news article lists from newsapi.org
/users/authenticate POST	front-end app	Used to authenticate users for login
/queries GET	front-end app	Used to manage the list of saved queries
/queries POST	front-end app	Used to persist the list of saved queries

The following endpoints are not required by the initial design but would most likely be needed to support additional requirements.

Endpoint	Purpose
/users GET	Allow direct access to user records via REST api
/users POST	Allow insert of new user records over REST api
/users PUT	Allow update of existing records over REST api
/users DELETE	Allow deletion of user records over REST api
/users/register POST	Allow registration of new users over REST api

Create the Project

This will involve several steps including:

- Create working and project directories
- Initialize and setup the project directory
- Create a server file
- Initialize a Git repository for the project
- Run the server and test the home page

1. Open a terminal
2. Navigate to the following working directory (create it if needed):

C:\ProjectWork

3. Create the following project directory:

C:\ProjectWork\back-end

4. Navigate into the project directory.
5. Initialize the project directory:

```
npm init -y
npm install express mongodb
```

6. Edit the package.json, replace the 'scripts' section with the one below:

```
"scripts": {
  "start": "node server.js",
  "mongodb": "C:/Software/MongoDB/Server/6.0/bin/mongod",
  "mongosh": "C:/Software/mongosh/mongosh-1.8.0-win32-x64/bin/mongosh"
},
```

7. Add the following (in bold) to the package.json file. This sets the project type to "module" which allows us to use import statements to more easily connect code from multiple JavaScript files.

```
"version": "1.0.0",  
"type": "module",  
"description": "",
```

8. Save the **package.json** file.

Initialize the Git Repository

It will help if we can back out of any trouble we get ourselves into during implementation. This can be done by periodically committing changes to Git. For this to work we need first to initialize a Git repository in the server directory.

9. Create a new file in the project directory named **".gitignore"**
10. Add the following contents into the **.gitignore** file and save it:

```
node_modules  
package-lock.json  
queries.json
```

11. Go to the command prompt.
12. Make sure you are in the project directory: **C:\ProjectWork\back-end**
13. Run the following statements to initialize the Git repository and complete your first commit:

```
git init  
git add .  
git commit -m "initial commit"
```

Create the Server

For this server implementation you will separate the various routers out into their own files. Organizing the code makes it easier to work with.

14. Create a "routes" directory inside the server project directory
15. Create the following directories (in bold) inside the "routes" directory

```
routes\news  
routes\queries  
routes\users
```

16. Create a basic router file within each of the routes subdirectories.

```
routes\news\news.js  
routes\queries\queries.js  
routes\users\users.js
```

17. Edit each of the empty router files. Add the contents of the **ProjectAssets\back-end\router-init.txt** to each of the files and save them.

18. Create a file in the root of the project named: **server.js**

19. Add code to the server.js file that:

- imports express
 - imports path
 - creates an express app object
 - sets the port to 4000
 - calls app.use() to make use of the express.json() middleware
 - import the following router implementations:
 - newsRouter from \routes\news.js
 - queriesRouter from \routes\queries.js
 - usersRouter from \routers\users\users.js
 - add app.use() calls to integrate the following router implementations:
 - \routes\news\newsRouter
 - \routes\queries\queriesRouter
 - \routers\users\usersRouter
 - use app.listen(...) to start the server
- { see ProjectAssets\back-end\server-init.txt file for an example code implementation }

20. Save **server.js**

21. Open a command prompt and navigate to the "back-end" project directory.

22. Execute the following statement to start the server. If you receive any errors, fix them and try again:

```
npm run start
```

23. You should see the following:

```
> back-end@1.0.0 start
```

```
> node server.js  
  
Server listening on port 4000
```

24. Open the Postman REST client.

25. Try making requests to the following endpoints:

Endpoint & Request Type	Expected Response
http://localhost:4000/news GET	Status: 200 "GET called"
http://localhost:4000/news POST	Status: 200 "POST called"
http://localhost:4000/users GET	Status: 200 "GET called"
http://localhost:4000/users POST	Status: 200 "POST called"
http://localhost:4000/queries GET	Status: 200 "GET called"
http://localhost:4000/queries POST	Status: 200 "POST called"

26. Fix any errors, restart the server and confirm that the endpoints are working.

As you may have noticed, the current endpoint responses are generic. As you work through stage01 you will add all the code that's needed to complete the endpoints.

27. In your notes, document the current state of the repository:

```
state: "initial server with dummy endpoints"  
tag: "stage02-01"
```

28. Add a tag so that you can return to this state.

```
git tag "stage02-01"
```

Complete the "/news GET" endpoint

The /news GET endpoint will be used to verify that the API is up and running. It will call the newsapi.org REST API with a simple hard-coded query. We will need the following to implement this endpoint:

- A hard-coded query object
- An apiKey
- The baseUrl of newsapi.org's "top-headlines" endpoint
- A way to convert the hard-coded query to a query string.

We are going to use JSON format to pass queries from the front-end client application to this "back-end" server. The client will pass a queryObject in the body of the "back-end" server request. For our purposes with this endpoint we will just add the queryObject as a variable inside the GET handler.

1. Edit the /routes/news/news.js file.
2. Add the following as the first line in the "router.get(...)" statement:

```
let fixedQueryObject = {  
  "country": "us",  
  "q": "news"  
}
```

Before we can use the queryObject we need to add apiKey you got after registering with newsapi.org.

3. Add the following function to news.js right after the "const router ..." line:

```
const apiKey = process.env.API_KEY;  
if(!apiKey){  
  console.log("Please set the API_KEY environment variable with a valid  
newsapi.org apiKey and restart the server!");  
  process.exit(0);  
}
```

This code:

- Creates a variable to hold the apiKey
- Sets the apiKey value from the API_KEY environment variable
- Exits the server app if it is not set.

The code above can be copied from the file: **ProjectAssets\back-end\api-key-code.txt**

4. Add the following function to news.js right before the GET handler method. This method will take a query object and add the apiKey to it.

```
function addApiKey(queryObject){  
  return {...queryObject, apiKey: apiKey}  
}
```

This function can be copied from the file: **ProjectAssets\back-end\add-api-key.txt**

5. Add the following line of code (shown in bold) after the fixedQueryObject definition:

```
let fixedQueryObject = {  
  "country": "us",  
  "q": "news"  
}  
let queryObject = addApiKey(fixedQueryObject);
```

```
res.send(`GET called`);
```

Before we can make the call to the restapi.org API we still need to create the URL. The URL is made up of the baseUrl for restapi.org's "top headlines" endpoint, plus a query string created from the queryObject.

6. Add the following line of code (shown in bold) just before the "function addApiKey ..." line:

```
const baseUrl = 'https://newsapi.org/v2/top-headlines';
```

Once the query is in the back-end it must be converted to a query string. This can be done using the JavaScript method "URLSearchParams".

7. Add the following function to new.js just after the "function addApiKey(){..}" method:

```
export function createUrlFromQueryObject(queryObjectWithApiKey) {  
  const queryString = new  
  URLSearchParams(queryObjectWithApiKey).toString();  
  const url = baseUrlTop + "?" + queryString;  
  return url;  
}
```

This function can be copied from: **ProjectAssets\back-end\create-url-function.txt**

8. Add the following function just after the "function createUrlFromQueryObject(){...}" method:

```
export async function fetchData(url) {  
  let data = null;  
  try {  
    const response = await fetch(url);  
    if (!response.ok) {  
      throw new Error(`HTTP error! status: ${response.status}`);  
    }  
    data = await response.json();  
    return data;  
  } catch (error) {  
    console.error('Error fetching data:', error);  
    return null;  
  }  
}
```

This function can be copied from: **ProjectAssets\back-end\get-data-function.txt**

9. Edit the "router.get() ..." route handler method. Replace the existing "res.send(...)" statement with the following code:

```
let url = createUrlFromQueryObject(queryObject);  
let newsArticles = await fetchData(url);
```

```
res.send(newsArticles);
```

The completed "router.get()" handler implementation is available here: **ProjectAssets\back-end\complete-news-get.txt**

10. Save the news.js file.
11. Stop the back-end server with Ctrl-C, Y {enter}
12. Start the server again using: npm run start
13. You should see the following error:

```
Please set the API_KEY environment variable with a valid newsapi.org apiKey  
and restart the server!
```

14. On the command line, set the API_KEY using the value you received when you registered with newsapi.org. The command should look something like this:

```
Syntax: set API_KEY={your-api-key}  
Example: set API_KEY=b24f9d5564944ab3acd556097bece85c
```

15. After setting the API_KEY environment variable, restart the server: npm run start
16. Using a browser, try running a GET query against the /news endpoint

<http://localhost:4000/news>

You should get a list of articles.

17. Try making the same request using Postman, you should get a similar result.



18. If any errors come up, fix them and try again to call the news endpoint as described above. Once this is working, you should note the current state of the code as:

```
state: "news GET endpoint working"  
tag: "stage02-02"
```


19. Commit and tag your code so that you can return to this state.

```
git add .
git commit -m "news GET endpoint is working"
git tag "stage02-02"
```

Complete the **"/news POST"** endpoint

The /news POST endpoint accepts queryObjects from the front-end application, calls the newsapi.org REST API and returns the results. The only difference between this endpoint and the /news GET endpoint you already created is where the queryObject comes from.

20. Edit the /routes/news/news.js file.

21. Copy the contents of the "router.get()" handler call over to the "router.post()" handler call.

At this point calling the GET endpoint and the POST endpoint would return the same data.

22. Make the following changes to the POST handler:

- Delete the "fixedQueryObject"
- Add the following as the first line in the handler: `const query = req.body`
- Adjust the rest of the code to use the request body query instead of the former fixedQueryObject

The result should look something like this:

```
router.post('/', async (req, res) => {
  let query = req.body;
  let queryObjectWithApiKey = addApiKey(query);
  let url = createUrlFromQueryObject(queryObjectWithApiKey);
  let newsArticles = await fetchData(url);
  res.send(newsArticles);
});
```

The code above is available in the file: ProjectAssets\back-end\complete-news-post.txt

23. Save the news.js file.

24. Stop the back-end server with Ctrl-C, Y {enter}

25. Start the server again using: npm run start

26. Using Postman, create a post request with the following qualifications:

- URL: `http://localhost:4000/news`
- Type: POST
- Body Type: raw JSON

- Body Content:

```
{  
  "country": "us",  
  "pageSize": 3,  
  "q": ""  
}
```

This queryObject will bring back the first three top news articles. Notice that the query itself ('q') is set to the empty string. This means that the overall list retrieved, which is represented by the "totalResults" property in the response, will not be restricted. The pageSize of 3 though asks the newsapi to return just the first three items out of the total results.

27. Execute the POST request and note the result. The result consists of an object like this:

```
{  
  "status": "ok",  
  "totalResults": 36,  
  "articles": [ {...}, {...}, {...}]  
}
```

The front-end client you create later will consume this object type and display the articles in a list.

28. Once you've got the endpoint working, Clean up your code and Commit your changes and Tag your progress:

```
git add .  
git commit -m "news POST endpoint working"  
git tag stage02-03
```

Create "/users GET" endpoint

One of the front-end application's requirements is that users be able to login to the application. This will be done by checking their credentials (user/pass) against records saved in the MongoDB usersdb users collection.

The "/users GET" endpoint provides a way to check which users are in the database without having direct access to the database itself.

You created the users collection in MongoDB when you setup the demo application. We will have this endpoint get its data from there.

The authentication of credentials will be done by the endpoint we create after this one - the "/users/authenticate POST" endpoint.

1. Create the following file:

```
\routes\users\db.js
```

2. Copy the contents of `ProjectAssets\back-end\mongodb-connect.txt` into `db.js` and save the file.

The code you just copied should look like this:

```
import {MongoClient} from "mongodb";
const connectionString = 'mongodb://127.0.0.1:27017';
export const client = new MongoClient(connectionString);

export async function connect() {
  try {
    await client.connect();
    console.log('MongoDB connected');
  } catch (err) {
    console.error(err);
  }
}
```

A connection to the client is made by calling "connect()"

3. Edit `/routes/users/users.js`
4. Add the following code right after the imports at the top of the file:

```
import {client, connect} from './db.js';
const dbName = 'usersdb';
const collectionName = 'users';
connect(); // Connect to MongoDB
```

The above code:

- Imports client and connect from `db.js`, these will be used to connect to the db and to access the mongodb client
- Sets up db and collection name constants that are used for creating mongodb queries
- runs the connect method to connect to mongodb

5. Replace the contents of the "router.get()" handler with the following code:

```
try {
  const db = client.db(dbName);
  const collection = db.collection(collectionName);
  const prj = {user:1,email:1, _id:0};
  const users = await collection.find({}).project(prj).toArray();
  res.json(users);
} catch (err) {
  res.status(500).json({ error: err.message });
}
```

The above code:

- Sets up the `db.collection` which is later used to query mongodb

- The prj variable holds an array that determines which data columns(properties) from the collection we want returned.
- We use "find()" to return all rows and just the selected columns. For security reasons we don't want to return the password field.
- The res.json(users) converts the array into a json string and returns it to the client
- If any errors occur we return the error

A copy of the completed GET handler can be found at: **ProjectAssets\back-end\complete-users-get.txt**

6. Save users.js
7. Stop and restart the server. (CtrlC, Y), (npm run start)

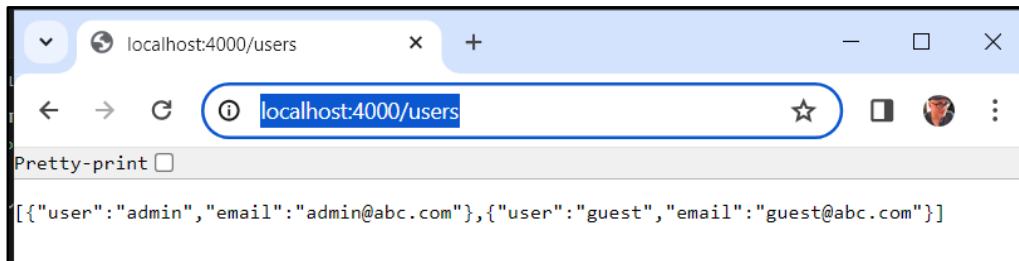
If mongodb is not running you will get an error the first time you call the "/users GET" endpoint. Make sure that mongodb is running. It can be started from the project directory with:

```
npm run mongodb
```

8. Make a GET request to the following URL(use Postman, or browser):

<http://localhost:4000/users>

9. You should get the two records you added when setting up the demo



10. Once you've got the GET endpoint working, Clean up your code and Commit your changes and Tag your progress:

```
git add .  
git commit -m "users GET endpoint working"  
git tag stage02-04
```

Create "/users/authenticate POST" endpoint

The "/Users/authenticate POST" endpoint will be called by the front end client to check if the credentials entered by a user match those in the MongoDB database. If they match then the endpoint should return status 200-OK, if not then the endpoint should return 401-unauthorized.

To code the endpoint you will need to:

- Set the path for the "router.post" call to "/authenticate
- Get a MongoDB collection object for the "users" collection
- Get the loginCredentials from the request body
- Use the collection.findOne() method to retrieve a record with the same name as is presented in the loginCredentials.
- If the retrieved record is null then the user does not exist in the database and authentication should be denied (status 401
- If the retrieved record IS returned then its password should be compared with the password in the loginCredentials.
- If passwords match then you should return status 200-OK to confirm authentication
- If passwords don't match then you should return status 401

Example code implementing the "/users/authenticate POST" endpoint can be found at:

ProjectAssets\back-end\complete-users-post.txt

11. Save users.js

12. Using Postman, create a POST request with the following qualifications:

- URL: http://localhost:4000/users/authenticate
- Type: POST
- Body Type: raw JSON
- Body Content:

```
{  
  "user": "admin",  
  "password": "admin"  
}
```

13. Execute the POST request and note the result. You should get a status 200 and the response body should include the message: "user passed authentication!"

14. Change the the **user** to "adminx" and try the POST again. This time you should get a status 401.

15. Change the the **user** back to "admin" and change the **password** to "adminx" and try the POST again. You should get another status 401.
16. Once you've got the **"/users/authenticate POST"** endpoint working, Clean up your code, commit your changes and tag your progress:

```
git add .  
git commit -m "users/authenticate POST endpoint working"  
git tag stage02-05
```

Create **"/queries POST/GET"** endpoints

When a user submits a query in the front-end application the query is used to retrieve a list of articles from the newsapi.org REST API and then the query is added to a list of saved queries that is displayed on-screen.

The job of the **"/queries"** endpoint of our back-end REST API is to persist the saved query list. Everytime a new query is added to the list in the front-end the entire list will be saved using the **"/queries POST"** endpoint. Then if the user exits the front-end application in the browser and starts it up again the **"/queries GET"** endpoint will be called to retrieve the saved query list.

Although it's possible to save the list in the MongoDB database, we are instead going to persist this list to a JSON formatted file on the REST server.

1. Edit the routes\queries\queries.js file
2. Add the following import at the top of the file:

```
import fs from 'fs';
```

This import will allow us to access the server's file system.

3. Add code to the "router.post()" handler that does the following:
 - Gets the saved query array from the incoming request
 - Use JSON.stringify to convert the array to a string.
 - Use fs.writeFileSync() to save the string version of the array to a file named: "queries.json"
 - To catch any failures you should wrap the writeFileSync with try{}catch{}
 - If writeFileSync fails then return status 500
 - If writeFileSync succeeds then return the message "query array saved" along with the status of 200.

Example code implementing the **"/queries POST"** endpoint can be found at: **ProjectAssets\back-end\complete-queries-post.txt**

4. Save queries.js
5. Stop and restart the server
6. Using Postman, create a POST request with the following qualifications:
 - URL: `http://localhost:4000/queries`
 - Type: POST
 - Body Type: raw JSON
 - Body Content: copy from `ProjectAssets\back-end\queries-json.txt`

The Body Content should look like this:

```
[
  {
    "queryName": "query01",
    "language": "en",
    "pageSize": 10,
    "q": "news"
  },
  {
    "queryName": "query02",
    "language": "en",
    "pageSize": 10,
    "q": "build"
  }
]
```

7. Execute the `/queries POST` request. You should receive a status 200 and the message "query array saved". If not then fix the code and try the POST again.
8. After executing the POST and getting a status 200 take a look in the root directory of your project. the following file should now appear:

`queries.json`

This file was saved the by the server into the directory where the server was started from. Check the file's contents. It should be the same as what you posted in the request body.

9. Edit `routes\queries\queries.js`.
10. Update the `"router.get()"` handler to do the following:
 - Read the contents of the 'queries.json' file into a variable using `fs.readFileSync`
 - Wrap `readFileSync` with a `try{}catch{}` so that you can catch any possible read errors
 - If the read fails then return a status 404-not found
 - If the read succeeds then return the data that was read from the file and a status of 200.

Example code implementing the **"/queries GET"** endpoint can be found at: **ProjectAssets\back-end\complete-queries-get.txt**

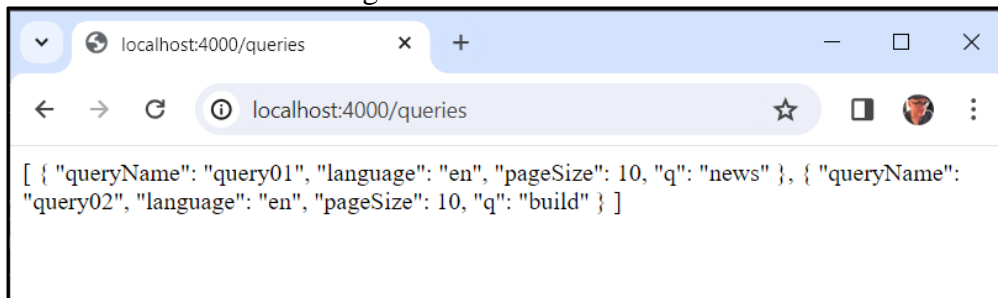
11. Save queries.js

12. Stop and restart the server.

13. Using a browser execute a GET request with the following URL:

```
http://localhost:4000/queries
```

You should see the following:



The query returned the contents of the "queries.json" file on the server.
Executing the same GET from within Postman should give you similar results.

14. Once you've got the **"/queries POST"** and the **"/queries GET"** endpoints working, Clean up your code, commit your changes and tag your progress:

```
git add .  
git commit -m "queries POST & GET endpoints working"  
git tag stage02-06
```

Your server now has everything it needs to support the front-end client's base requirements!

Stage03 – Create the "front-end" React Client Application

In this stage you will:

- Setup the front-end-starter application
- Update the app to get live data from back-end server
- Add the SavedQueries list component
- Update the SavedQueries list to persist after a restart
- Add login capabilities
- Add login restricted features

Setup the front-end starter application

1. Copy front-end-starter.zip from ProjectAssets to your working directory C:\ProjectWork
2. Right click on the zip file and choose "Extract All..."
3. When the Extract dialog appears choose all defaults and click on "Extract"
4. The directory "front-end-starter" will be created
5. Rename the directory to "front-end"
6. Open a command prompt
7. Navigate to the ProjectWork\front-end directory
8. Execute the following to install dependencies:

```
npm install
```

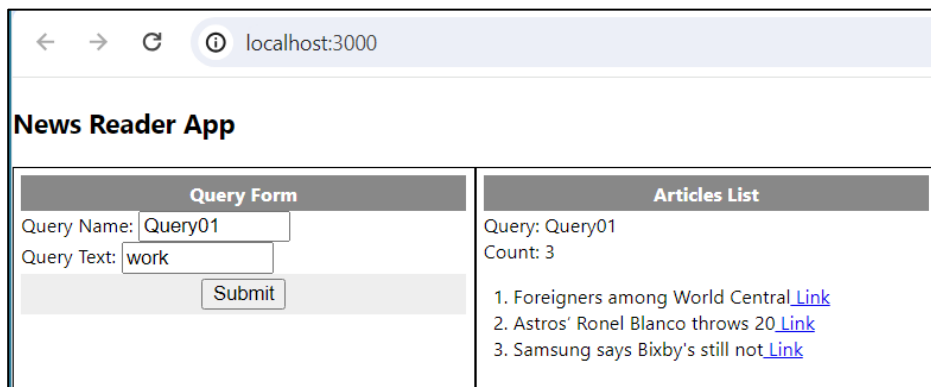
9. Execute the following to start the front-end application:

```
npm run start
```

10. Open a browser to this URL:

<http://localhost:3000/>

11. You should see this:



Query Form	Articles List
Query Name: <input type="text" value="Query01"/> Query Text: <input type="text" value="work"/> <input type="button" value="Submit"/>	Query: Query01 Count: 3 1. Foreigners among World Central Link 2. Astros' Ronel Blanco throws 20 Link 3. Samsung says Bixby's still not Link

The query and article data you see is currently hard coded into the application.

12. The application consists of the following Components:
 - App
 - NewsReader
 - QueryForm
 - Articles

13. Take some time and review how the application's components work and how they fit together. You'll need to understand how the application is structured in order to build it out further.
14. Open up a second terminal and navigate to the front-end project directory
15. Initialize a git repository for the front-end project:

```
git init
git add .
git commit -m "starting point"
git tag stage03-01
```

Update Application to use the server's "/news" endpoint

1. Before updating the application we need to make sure that:
 - MongoDB is running
 - The front-end server is running
2. If they are not already running, Start the servers:
 - Open a command prompt and navigate to the **ProjectWork\back-end** directory.
 - Start the MongoDB server by executing: **npm run mongodb**
 - Open another command prompt and navigate to the **ProjectWork\back-end** directory.
 - Set the API_KEY env variable: **set API_KEY={your-newsapi.org-api-key}**
 - Start the back-end server by executing: **npm run start**
 - Check the server by calling this url in a browser: **http://localhost:4000/news**

Now we can start working on the update

3. Edit **src\NewsReader.js**
4. At the top of the file you will find this import statement:

```
import { exampleQuery ,exampleData } from './data';
```

The exampleQuery and exampleData point to the sample data that is currently populating the QueryForm and Articles components.

5. Find the getNews() function.

```
async function getNews(queryObject) {
  if (queryObject.q) {
    setData(exampleData);
  }
}
```

```
    } else {  
      setData({});  
    }  
  }  
}
```

This is where the sample data is being used. We want to update the `getNews()` function so that it gets its data from the `"/news"` endpoint of the back-end REST server.

6. Add the following code inside the `NewsReader()` function, after the `useState` statements:

```
const urlNews="/news"
```

7. Update the `getNews()` function so that:

- When `queryObject.q` is null or empty `setData()` is called with an empty object
- When `queryObject.q` is not empty, make a `fetch()` request and pass the `queryObject` as the request body (don't forget to stringify it)
- Throw an error if the fetch response does not come back ok
- Call `response.json()` to get the response body (the article data)
- Pass the response data to `setData()`

An example implementation of `getNews()` is available in the file: **ProjectAssets\front-end\get-news.txt**

8. Save `NewsReader.js`

9. Refresh the browser

10. Change the "Query Text" field value to "state"

11. Click on the "Submit" button

12. You should see the articles list change. If not articles show up then try a different search term for "Query Text"

13. Try a few other search terms, remember to click submit after each change

14. Once the new code is working, commit and tag your progress:

```
git add .  
git commit -m "now gets live article list from server"  
git tag stage03-02
```

Add SavedQueries Feature

The `SavedQueries` feature will allow application users to name, save and recall queries that they submit.

Changes needed to support this feature include:

- Add a variable in `NewReader` to hold the saved query list

- Update the list whenever a new query is submitted
- Add a React component to display the query list
- Clicking on an item in the saved query list should set the current query
- When the list is updated, save it to the back-end server
- Load the list when the front-end client starts up

1. Edit NewsReader.js
2. Add the following code after the "const urlNews..." statement:

```
const [savedQueries, setSavedQueries] = useState([...exampleQuery ]]);
```

3. Update the "onFormSubmit()" method to include the code shown in bold below:

```
function onFormSubmit(queryObject) {  
  let newSavedQueries = [];  
  newSavedQueries.push(queryObject);  
  for (let query of savedQueries) {  
    if (query.queryName !== queryObject.queryName) {  
      newSavedQueries.push(query);  
    }  
  }  
  console.log(JSON.stringify(newSavedQueries));  
  setSavedQueries(newSavedQueries);  
  setQuery(queryObject);  
}
```

Code for the above onFormSubmit () method is available in the file: **ProjectAssets\front-end\on-form-submit01.txt**

Notice the console.log() statement in the above code. It will display the updated saved query list in the browser's JavaScript console. Its output can be used to check if the list is being updated properly.

4. Save the **NewsReader.js** file

Next we will create a component that can be used to display the list.

5. Create a new file in the front-end project:

```
src\SavedQueries.js
```

6. Copy the contents of **ProjectAssets\front-end\saved-queries.txt** into the newly created file.
7. Save the **SavedQueries.js** file
8. Edit the **NewsReader.js** file
9. Add the following import statement at the top of the file:

```
import { SavedQueries } from './SavedQueries';
```

10. Update the JSX code in the return statement as shown in bold below:

```
<div className="box">
  <span className='title'>Query Form</span>
  <QueryForm
    setFormObject={setQueryFormObject}
    formObject={queryFormObject}
    submitToParent={onFormSubmit} />
</div>
<div className="box">
  <span className='title'>Saved Queries</span>
  <SavedQueries savedQueries={savedQueries}
    selectedQueryName={query.queryName} />
</div>
<div className="box">
  <span className='title'>Articles List</span>
  <Articles query={query} data={data} />
</div>
```

11. Save the **NewsReader.js** file

12. Refresh the application in the browser. You should now see something like this, though the articles in the list may differ.

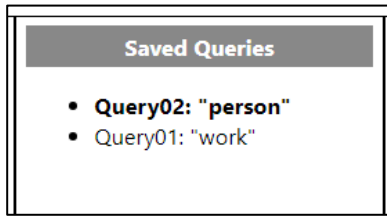
News Reader App		
Query Form	Saved Queries	Articles List
Query Name: <input type="text" value="Query01"/> Query Text: <input type="text" value="work"/> <input type="button" value="Submit"/>	<ul style="list-style-type: none">Query01: "work"	Query: Query01 Count: 3 1. World Central Kitchen halts op Link 2. Foreign Aid Workers From Jose Link 3. California's New \$20 Fast-Food Link

13. Change the "Query Name" in the Query Form to "Query02"

14. Change the "Query Text" in the Query form to "person"

15. Click "Submit"

16. The "Saved Query" List should update to include the new query:



Currently, if you click on an item in the Saved Query list it will throw an exception. We need to add a function that updates the current query based on an item selected in the Saved Query list.

17. Edit the **NewsReader.js** file

18. Add the following method before the "onFormSubmit()" method:

```
function onSavedQuerySelect(selectedQuery) {  
    setQueryFormObject(selectedQuery) ;  
    setQuery(selectedQuery) ;  
}
```

Code for the above method is available in the file: **ProjectAssets\front-end\on-saved-query-select.txt**

19. Update the `<SavedQuery ...>` element in the return statement as shown below in bold:

```
<SavedQueries  
    savedQueries={savedQueries}  
    selectedQueryName={query.queryName}  
    onQuerySelect={onSavedQuerySelect} />
```

20. Save the **NewsReader.js** file

21. Refresh the front-end application in the browser.

22. Enter the following query and click submit:

```
Name: Query02  
Text: person
```

23. The new query will be added to and highlighted in the Saved Query list and the articles list will be updated.

24. Click on "Query01" in the Saved Queries list. The clicked on query will be highlighted and the articles list will be updated.

25. Try submitting a few more queries and clicking on the different queries in the Saved Queries list.

26. Once the new code is working, commit and tag your progress:

```
git add .  
git commit -m "added SavedQueries list"  
git tag stage03-03
```

So far so good except that when you refresh the browser the list is lost. In the next part you will make use of the back-end server's "queries" endpoints to persist the SavedQueries list.

Persisting the SavedQueries List

To persist the saved queries list you will need to:

- Add a method in NewsReader.js that posts the updated queries list to the back-end server
- Add a method that gets the saved queries list from the back-end server
- Add code that calls the get method when front-end starts up
- Add code that calls the post/save method when the queries list is updated

1. Edit the **NewsReader.js** file
2. Add the following statement after the "const urlNews ..." statement:

```
const urlQueries = "/queries"
```

3. Add the following two methods before the "function onSavedQuerySelect..." function:

```
async function getQueryList() {
  try {
    const response = await fetch(urlQueries);
    if (response.ok) {
      const data = await response.json();
      console.log("savedQueries has been retrieved: ");
      setSavedQueries(data);
    }
  } catch (error) {
    console.error('Error fetching news:', error);
  }
}

async function saveQueryList(savedQueries) {
  try {
    const response = await fetch(urlQueries, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(savedQueries),
    });

    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }
    console.log("savedQueries array has been persisted:");
  } catch (error) {
    console.error('Error fetching news:', error);
  }
}
```

Code for the above methods is available in the file: **ProjectAssets\front-end\persist-query-list.txt**

4. Add the following `useEffect` after the `getNews` `useEffect` statement in `NewReader`:

```
useEffect(() => {getQueryList();}, [])
```

5. Update the `onFormSubmit()` method to add the statement shown in bold below:

```
function onFormSubmit(queryObject) {  
  let newSavedQueries = [];  
  newSavedQueries.push(queryObject);  
  for (let query of savedQueries) {  
    if (query.queryName !== queryObject.queryName) {  
      newSavedQueries.push(query);  
    }  
  }  
  console.log(JSON.stringify(newSavedQueries));  
  saveQueryList(newSavedQueries);  
  setSavedQueries(newSavedQueries);  
  setQuery(queryObject);  
}
```

6. Save the **NewsReader.js** file

Next you will test to make sure that the list is being persisted.

7. Delete the following file:

```
ProjectWork\back-end\queries.json
```

Deleting this file resets the list that the server is persisting.
Its OK if the back-end server is running when you delete the file.

8. Refresh the front-end application in the browser.

A single query "Query01" should appear in the query list.

9. Create a new query named "Query02" with the query text "person" and click "Submit"

Two queries now appear in the list, "Query01" and "Query02".

10. Refresh the front-end application again.

When the application has finished loading the list should still contain the two queries. (before this update it would have gone back to having just the single query)

11. Try adding a few more queries and then refreshing the browser again. You should see that all the queries you create remain in the application even after refresh.
12. Once the update is working, commit and tag your progress:

```
git add .
git commit -m "SavedQueries are now retained after refresh"
git tag stage03-04
```

Add User Login Capabilities

All of the te application's features are currently available to any user of the site. By adding login capabilities you will later be able to setup the site with features that only certain users can access. The back-end server includes a **users/authenticate** endpoint that can be used to authenticate users.

Adding login capabilities to the front-end application will involve:

- Adding a variable to hold user credentials
- Adding a variable to hold the current logged-in user
- Adding a "login" method that authenticates credentials by calling the the back-end server
- Adding a UI component that
 - Displays the user login state
 - Includes input fields to accept the users credentials
 - Includes a "login/logout" button
- ...

1. Edit the **NewsReader.js** file
2. Add the following useState variables after the other useState variables near the top of the file:

```
const [currentUser, setCurrentUser] = useState(null);
const [credentials, setCredentials] = useState({ user: "", password: "" });
```

3. Add the following variable after the "const urlQueries ..." statement:

```
const urlUsersAuth = "/users/authenticate";
```

4. Add a "login" function after the "getQueryList()" useEffect statement. This function will double as both a login and logout function.
5. Code the following into the login() function:
 - If login is called when there is valid currentUser then call setCurrentUser(null) to logout
 - If login is called when currentUser is null then authenticate and login
 - To authenticate, make a POST request of the urlUsersAuth url and pass the current credentials object in the request body.
 - If response.status is 200 then:

- setCurrentUser from the credentials object
 - unset the credentials user and password values by setting them to ""
- if the response.status is NOT 200 then:
 - Post an alert("err during authentication, update credentials and try again")
 - setCurrentUser(null)

Code for the "login" function is available in the file: **ProjectAssets\front-end\login-function.txt**

6. Save the **NewsReader.js** file
7. Create a file in the src directory named "**LoginForm.js**"
8. Copy the contents of ... into the login form.
9. Save the LoginForm.js file
10. Edit the **NewsReader.js** file
11. Add the following import at the top of the file after the other imports:

```
import { LoginForm } from './LoginForm';
```

12. Update the JSX code in the return statement as shown in bold below:

```
<div>  
  <LoginForm login={login}  
    credentials={credentials}  
    currentUser={currentUser}  
    setCredentials={setCredentials} />  
  <div >  
    <section className="parent" >
```

13. Save the **NewsReader.js** file
14. Refresh the front-end application in the browser
15. The login component should appear like this:

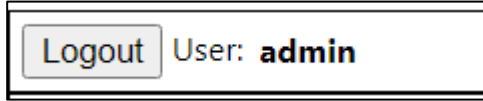


The screenshot shows a web application interface with a login form. At the top left is a button labeled "Login". To its right is the text "User: not logged in". Below this, there are two input fields: one labeled "User:" and another labeled "Password:". The "User:" field is currently empty, and the "Password:" field is also empty.

16. Enter the following credentials and click "Login":

```
User: admin  
Password: admin
```

17. The login component should change to this:



18. Clicking "Logout" should return the component to its previous state.

At this point the `currentUser` variable is available in the `NewsReader` component and can be used to allow or prohibit users from features as needed. The `currentUser`'s value is null when no one is logged in. When someone is logged in the `currentUser` contains an object with their credentials.

19. When you have the login feature complete and working, make sure to commit and tag your progress:

```
git add .
git commit -m "Login capability is now available"
git tag stage03-05
```

Add a Login Restricted Features

Based on the contents of our users database in mongodb and the implementation of our login feature these three states are now possible:

- No user logged in
- Guest user logged in
- Admin user logged in

In this part we will use the login states to enable/disable features as listed here:

State	currentUser	
No user logged in	null	Can execute any existing saved queries. No new queries can be submitted/created. An alert should appear if the user tries to submit/create any additional queries.
Guest logged in	{ user: "guest", password: "guest" }	Guest user has limited query submit/create abilities. Once three or more queries appear in the saved queries list, no more can be created. Guest is able to execute any existing saved queries. An alert should appear if guest tries to submit any new queries.
Admin logged in	{ user: "admin", password: "admin" }	Admin has access to additional query parameters when creating queries (page, searchIn) Admin can create an unlimited number of queries.

We will start by adding the page and sortBy query parameters. This change will involve:

- Adding page & sortBy to:
 - the exampleQuery object
 - the QueryForm
- Restricting access to the new attributes visually in the QueryForm

1. Edit the **data.js** file
2. Check the exampleQuery object to make sure it includes "language" and "pageSize":

```
export const exampleQuery = {  
  "queryName": "Query01",  
  "q": "work",  
  "language": "en",  
  "pageSize": 10  
};
```

3. Edit the **NewsReader.js** file
4. Add the following (in bold) to the JSX code for the QueryForm component:

```
<QueryForm  
  currentUser={currentUser}  
  setFormObject={setQueryFormObject}  
  formObject={queryFormObject}  
  submitToParent={onFormSubmit} />
```

5. Save the **NewsReader.js** file
6. Edit the **QueryForm.js** file.
7. Add the following function right before the return statement:

```
function currentUserIsAdmin() {  
  if (params.currentUser) {  
    if (params.currentUser.user) {  
      if (params.currentUser.user === "admin") {  
        return true;  
      }  
    }  
  }  
  return false;  
}
```

8. Add the following <div> to the JSX/HTML code in QueryForm. This section will be used to contain the extra query fields and allow conditional access to them:

```
<div className={ (currentUserIsAdmin()) ? "visible" : "hidden" }  
  style={{border: "solid black 1px"}}>  
  {/* Extra fields */}  
  <p>extra admin stuff</p>  
</div>
```

9. Replace this JSX/HTML code "**<p>extra admin stuff</p>**" from the code you just added with code that allows input of "language" and "pageSize" data.

Example code for this is in the file: **ProjectAssets\front-end\extra-query-inputs.txt**

10. Save the **QueryForm.js** file

11. Refresh the browser and login as admin/admin and verify that the additional fields appear in the query form.

Next we will implement these restriction on the guest user:

- Once three or more queries appear in the saved queries list, no more can be created.
- An alert should appear if guest tries to submit queries once three exist.
- When this happens the submit should be cancelled

To implement these restrictions you will need to make changes to the " onFormSubmit " function in the NewReader component.

12. Edit the **NewsReader.js** file

13. Add the following method just before the "onFormSubmit()" method:

```
function currentUserMatches(user) {  
  if (params.currentUser) {  
    if (params.currentUser.user) {  
      if (params.currentUser.user === user) {  
        return true;  
      }  
    }  
  }  
  return false;  
}
```

14. Add the following code, shown in bold to the "onFormSubmit()" method:

```
function onFormSubmit(queryObject) {  
  if (savedQueries.length >= 3 && currentUserMatches("guest")) {  
    alert("guest users cannot submit new queries once saved query count is 3  
or greater!")  
    return;  
  }  
  let newSavedQueries = []  
  ...
```

15. Save **NewsReader.js** file.

16. Refresh the front-end application.

17. If the number of saved queries is >3 already then delete the queries.json file in the back-end project directory and refresh the front-end app again.

18. Login as guest/guest

19. Add(submit) queries until you see the error alert pop up. Confirm that you cannot create more queries when logged in as guest.
20. Logout and login as admin/admin and verify that you can create (submit) queries even when the saved query count is over 3.

Next we will implement the restrictions that apply when no users are logged in:

- User can choose/submit any of the existing saved queries.
- An alert should appear if user tries to submit any new queries.
- When this happens the submit should be cancelled

To implement these restrictions you will again need to make changes to the " onFormSubmit " function in the NewReader component.

21. Edit the **NewsReader.js** file

22. Add the following code, shown in bold to the "onFormSubmit()" method:

```
function onFormSubmit(queryObject) {  
  if (currentUser === null){  
    alert("Log in if you want to create new queries!")  
    return;  
  }  
  if (savedQueries.length >= 3 && currentUserMatches("guest")) {
```

23. Save **NewsReader.js** file.

24. Refresh the front-end application.

25. If you are logged in then Logout.

26. Try submitting a new query from the Query Form. You should see an error alert pop up. Confirm that you cannot create any queries unless you are logged in.

27. Logout and login as admin/admin and verify that you can still create (submit) queries when logged in.

28. When this feature is complete and working, make sure to commit and tag your progress:

```
git add .  
git commit -m "Login restricted features added"  
git tag stage03-06
```

Stage04 – Add Optional Application Features

This stage is for those who have completed the earlier stages and still have some time left before they need to start preparing to present their capstone project. Its up to you which of the listed features you want to do.

In this stage you will update the application by adding several features. Unlike previous stages of the project, this stage will include less step-by-step instructions.

The idea is for you to use what you've learned about how the application works and how it is coded to decide for yourself the best way to implement the new feature. That being said feel free to reach out to other students or to the instructor and talk over any issues you come up with.

The following aspects of each feature will be provided:

- Feature title
- Feature description
- Base requirements list
- List of hints

Before coding a specific feature you should prepare by:

- Reading the provided feature documentation. Make sure you understand it.
- Create a list of requirements that outline the functionality you plan to implement
- Create a work-plan where you list the steps you intend to take in completing the implementation.
- Commit and tag the current state of each project you plan to work on (front-end, back-end) so you can roll-back any changes that are not working as planned.

While working:

- Commit changes often, but only when the application is in a working state.
- Do not commit broken or non-working code.
- Do not commit without meaningful commit messages that can be understood later when you need them.
- After completing a feature, Add a git tag to mark your progress

Features List

Number	Feature Area	Description
01	Articles List	Details of the current query should be displayed at the top of the article list
02	Articles List	Show more characters for each title in articles list
03	Articles List	Add ability to scroll the articles list
04	Account	Show "canned" query list when no user is logged in
05	Saved Queries	Add ability to reset the saved query list (delete all).
06	Saved Queries	Show details for item selected in saved query list

Features Details

Feature Number: 01

Title: Display query details in Articles component

Description: Details of the current query should be displayed at the top of the article list

Base Requirements:

- Area above articles list in Articles component should be created to hold details
- Styling of displayed details should not overshadow or pull the users attention too much away from the article list itself

Nice to have but not required:

- Allow user to toggle detail display on/off

Hints:

- Pass current query into the Articles component
- Create a function that formats the query properties into a displayable HTML list

Feature Number: 02

Title: Show more characters for each title in articles list

Description: Enhance the look of the article list by showing more of title text for each item

Base Requirements:

- Display more characters of the title
- Style the titles so that it is easier to read and does not take up too much space
- You should still be able to see a minimum of 5 list items without scrolling

Nice to have but not required:

- Remove the "link" and make the entire title text part of an anchor tag that links to the article.

Hints:

- Change the existing substring statement to allow for more characters
- Reduce the font-size for the title text
- Choose a font that is easier to read

Feature Number: 03

Title: Add ability to scroll the articles list

Description: The articles list should have a pre-defined height and width. Its height should not be determined by the number of articles being displayed. Instead, when the number of articles exceeds the pre-defined height a scroll-bar should be displayed that allows users to scroll down to the additional contents.

Base Requirements:

- Set the articles list to a predefined height
- A vertical scroll bar should be available to allow for scrolling of contents

Nice to have but not required:

- Scroll bar is hidden when the contents do not exceed the predefined height

Hints:

- Wrap the current list in a parent div
- Set a height for the parent div
- Set the CSS properties required to hide overflow content and to enable scrolling

Feature Number: 04

Title: Show a "canned" query list when no user is logged in

Description: By default, new queries cannot be created when a user is not logged in. But the app can still be used to run existing saved queries. When no user is logged in, Replace the saved query list with a default set of queries. This avoids having the list show queries that have been created or customized by an admin or guest user.

Base Requirements:

- Do not show customized query lists when no user is logged in
- Have a default set of queries that are displayed when no one is logged in
- When someone logs in, go back to showing the customizable list.

Nice to have but not required:

- Do not display the QueryForm unless there is a logged in user.

Hints:

- Log in as admin or guest and create a query list then copy the list items from the queries.json file (in the server project directory) and hard code them into the application to be used as the default list.
- Update the useEffect statement to load the default list instead of retrieving the customizable list from the back-end server.

Feature Number: 05

Title: Add ability to reset the saved query list

Description: The current application does not include a way to delete individual queries from the saved query list. This can be a problem for guest users who are restricted to creating only 3 queries and who cannot create any queries if someone else has already added three queries to the list. This feature would add the ability to erase the existing entries in the list. Though it doesn't go as far as allowing deletion of individual queries, the ability to clean out the list should be fairly easy and quick to implement.

Base Requirements:

- Add a "reset" button to the SavedQuery component
- Clicking the "reset" button should erase the currently saved queries and update the displayed list
- An alert "are you sure you want to erase the list?" should be popped up that allows the user to cancel the operation.

Nice to have but not required:

- Do not display the "reset" button when no user is logged in.

Hints:

- Create an empty list and use it to set the savedQuery variable
- Things might get complicated if the list is truly empty. If so then include the "exampleQuery" as a single query in the list that you set
- Don't forget to call saveQueryList and setSavedQueries with the updated array

Feature Number: 06

Title: Show details for item selected in saved query list

Description: Currently you enter your query in the QueryForm and then submit it. When it's submitted it gets included in the saved query list. But, not all the detail of the query is shown in the saved query list. In fact all you get currently is the query name and search term. Since admin users have several additional query properties they can set it would be nice if they could see them in the saved query list.

Base Requirements:

- Add a display area in the SavedQueries component for query properties
- When a query is selected, show its properties in the display area.

Nice to have but not required:

- Instead of only showing query properties for the selected item, update the list to display properties for each list item in the list itself. If you do this then think about displaying the query name in large bold letters and the query properties in a smaller font that is not bolded.

Hints:

- No hints on this one, just use HTML/CSS as usual

Stage05 – Clean up an Prepare for Presentation

At this point you need to get ready to present your work.

Regarding the presentation:

- You should understand what the application does so you can demonstrate it from its front-end GUI
- You should understand how the code supports each feature.
- You should understand how the front-end and back-end work together
- You should be able to speak about the work you've done, what worked, what didn't work, and what you learned along the way.

Work Steps:

- Stop implementing new features
- Get code ready to present. It should be free from bugs.
- If you are in the middle of an implementation and things are not yet working, rollback to latest commit where everything was working
- Clean-up code, remove console.log statements, remove unused code

- Re-test all of the app's features to verify they are working and if some are not working you should know that as well.
- Make a final commit "cleaned up code is ready for review".
- Don't forget to follow these steps for both the front-end and the back-end projects

Good luck with your presentation!

Where to go from Here

This section lists some ideas for features that might want to add to the newsreader application after this capstone project is completed.

As far as documentation goes, the ideas below all qualify as "advanced optional features" for which no hints or solution code will be provided. In fact the "ideas" are just that – basic ideas. The first task for those who are interested in delving into them would be to flesh out a set of requirements and work steps. What you come up with will be verified by your running tests against your completed code and by the end users who get to use the system.

Are you ready to try coding without a net? Why not give it a go?

Feature Enhancement Ideas:

Feature Area	Description	Complexity
Articles List	Include a paging control for the articles list	Medium
Query Form	Query Name should default to something like "myQuery01" where each time a new query is started the # is auto incremented.	Medium
Saved Queries	Add ability to delete individual saved queries	Medium
Server	Add apiKey requirement to back-end server endpoints	Medium
Account	Each logged-in user should have their own query lists	High
Server	Add endpoint access auditing to log who, what and when for endpoint requests	High